Neural Ordinary Differential Equations

Krishna Kumar

University of Texas at Austin

krishnak@utexas.edu

Overview

Learning Objectives

- Understand the connection between ResNets and continuous dynamics
- Master the Neural ODE framework and adjoint method
- Implement ODENets for image classification
- Apply continuous normalizing flows for generative modeling
- Build latent ODE models for irregular time series

Open Notebook

The ResNet Formula

A residual network transforms hidden states layer by layer:

$$h_{t+1} = h_t + f(h_t, \theta_t) \tag{1}$$

where $t \in \{0, 1, \ldots, T\}$ indexes the layers.

The Key Question

What happens as we add more layers ($T \to \infty$) and take smaller steps?

The Euler Connection

The ResNet update is the **Euler discretization** of an ODE:

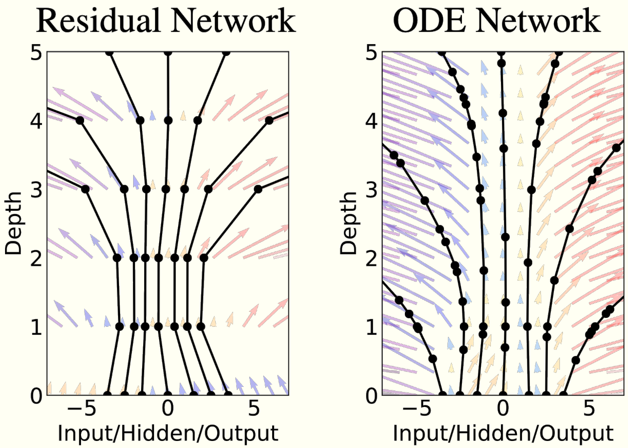
$$\frac{dh(t)}{dt} = f(h(t), t, \theta) \tag{2}$$

Key Insight

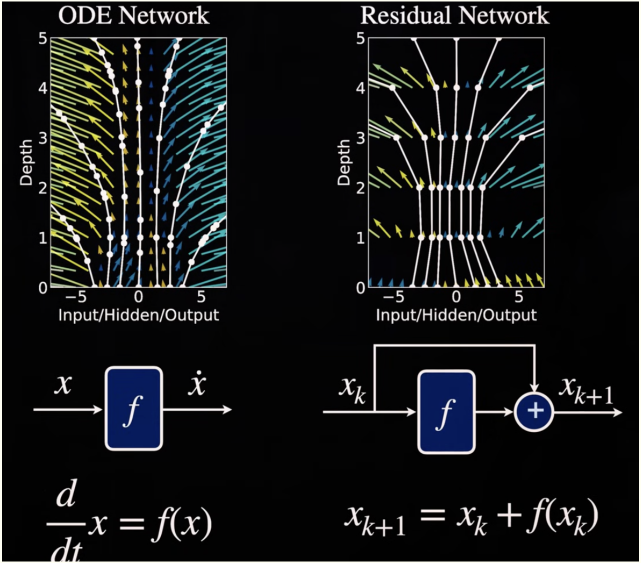
A ResNet with infinitely many infinitesimal layers $\equiv$ solving an ODE

Instead of specifying discrete layers, we parameterize the **derivative** of the hidden state using a neural network.

ResNet vs ODE



ResNet vs ODE



Numerical Integration: From Discrete to Continuous

The Mathematician's View:
The exact solution to $\frac{dx}{dt} = f(x, t, \theta)$ from time t_k to t_{k+1} is:

$$x_{k+1} = x_k + \int_{t=t_k}^{t=t_{k+1}} f(x(\tau), \tau, \theta)\, d\tau \tag{3}$$

The Problem: We can't compute this integral analytically!

The Solution: Numerical integration schemes approximate this integral

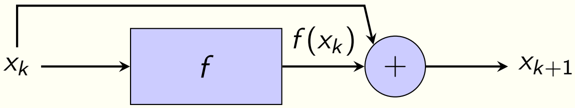
- **Euler (ResNet):** Simplest approximation, worst accuracy
- **Runge-Kutta:** Better approximations, higher accuracy
- **Adaptive methods:** Adjust step size automatically

Integration Schemes: Euler (ResNet)

Euler Method (Forward Euler):

$$x_{k+1} = x_k + \Delta t \cdot f(x_k, t_k, \theta) \tag{4}$$

With $\Delta t = 1$: $x_{k+1} = x_k + f(x_k, \theta) \quad \leftarrow$ ResNet!



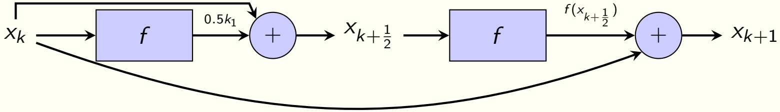
Properties:

- One function evaluation per step
- First-order accurate: error $\mathcal{O}(\Delta t^2)$
- Simple but can be unstable

Midpoint Method (2nd order Runge-Kutta):

$$k_1 = f(x_k, t_k, \theta) \tag{5}$$

$$x_{k+1} = x_k + \Delta t \cdot f(x_k + \frac{\Delta t}{2} k_1, t_k + \frac{\Delta t}{2}, \theta) \tag{6}$$



Properties:

- Two function evaluations per step
- Second-order accurate: error $\mathcal{O}(\Delta t^3)$
- Much more accurate than Euler for same Δt

Integration Schemes: RK4 (4th Order)

Classical 4th Order Runge-Kutta (RK4):

$$k_1 = f(x_k, t_k, \theta) \tag{7}$$

$$k_2 = f(x_k + \frac{\Delta t}{2} k_1, t_k + \frac{\Delta t}{2}, \theta) \tag{8}$$

$$k_3 = f(x_k + \frac{\Delta t}{2} k_2, t_k + \frac{\Delta t}{2}, \theta) \tag{9}$$

$$k_4 = f(x_k + \Delta t \cdot k_3, t_k + \Delta t, \theta) \tag{10}$$

$$x_{k+1} = x_k + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{11}$$

Properties:

- Four function evaluations per step
- Fourth-order accurate: error $\mathcal{O}(\Delta t^5)$
- Industry standard for non-stiff ODEs
- Much more stable than Euler/Midpoint

ResNet vs Neural ODE

Residual Network

- Discrete transformations
- Fixed number of layers T
- $x_{k+1} = x_k + f(x_k, \theta_k)$
- Memory: $\mathcal{O}(T)$
- Evenly spaced time steps

Neural ODE

- Continuous dynamics
- Adaptive depth
- $\frac{dx}{dt} = f(x(t), t, \theta)$
- Memory: $\mathcal{O}(1)$
- Irregular time steps OK

Basic Neural ODE Components

1. ODE Function $f(h, t, \theta)$

Neural network that computes the derivative $\frac{dh}{dt}$

2. ODE Solver

Integrates $h(t)$ from t_0 to t_1:

$$h(t_1) = h(t_0) + \int_{t_0}^{t_1} f(h(t), t, \theta)dt \tag{12}$$

3. Adjoint Method

Computes gradients $\frac{\partial L}{\partial \theta}$ efficiently

Memory Efficiency: The Adjoint Method

Standard Backpropagation:

- Store all intermediate layer activations
- Memory: $\mathcal{O}(L)$ where L = number of layers

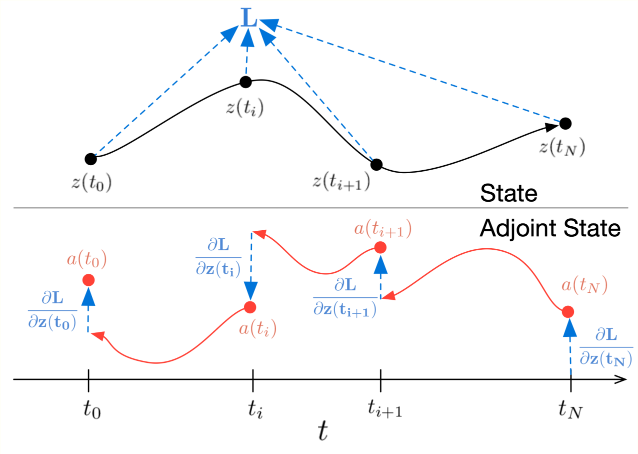
Adjoint Method:

- Solve a second ODE backwards in time
- Recompute forward states during backward pass
- Memory: $\mathcal{O}(1)$ independent of depth

The adjoint state $\lambda(t) = \frac{\partial L}{\partial h(t)}$ evolves as:

$$\frac{d\lambda(t)}{dt} = -\lambda(t)^T \frac{\partial f(h(t), t, \theta)}{\partial h} \tag{13}$$

The Adjoint Method Visualized



- **Forward**: Solve ODE from t_0 to t_1
- **Backward**: Solve augmented ODE from t_1 to t_0
- Automatically handled by `odeint_adjoint`

The Training Challenge

What we're doing: Learning the vector field $f(x, t, \theta)$

The Process:

1. Tweak parameters θ of the neural network
2. Numerically integrate along the vector field
3. Compare integrated trajectory to observed data
4. Minimize loss between prediction and data

Key Challenge: Hidden states between observations
Data is sampled at discrete times: $t_0, t_1, t_2, \ldots$
But the trajectory flows continuously: $x(\tau)$ for all $\tau \in [t_0, t_1]$

The Question

How do we compute gradients through the ODE solver without storing all intermediate states?

Step 1: A Concrete Constrained Problem

Goal: Minimize $f(x, y) = x^2 + y^2$ subject to $x + y = 1$

- Geometrically: find the closest point on the line $x + y = 1$ to the origin
- Naive elimination: substitute $y = 1 - x$ and minimize $g(x) = x^2 + (1 - x)^2$
- Differentiate $g'(x) = 4x - 2$, set to zero: $x = 1/2$ and $y = 1/2$
- Works in tiny problems, but elimination explodes with many constraints/variables

Observation

We need a reusable way to enforce constraints without eliminating variables one-by-one.

Step 2: Lagrange Multipliers Refresher

Introduce a multiplier λ and form

$$\mathcal{L}(x, y, \lambda) = x^2 + y^2 + \lambda(x + y - 1).$$

Stationarity $\nabla\mathcal{L} = 0$ gives

$$2x + \lambda = 0, \qquad 2y + \lambda = 0, \qquad x + y - 1 = 0.$$

- $\partial\mathcal{L}/\partial x = 2x + \lambda = 0$ (gradient in x direction)
- $\partial\mathcal{L}/\partial y = 2y + \lambda = 0$ (gradient in y direction)
- $\partial\mathcal{L}/\partial\lambda = x + y - 1 = 0$ (enforces the constraint)

1. From $2x + \lambda = 0$ and $2y + \lambda = 0$ we get $x = -\lambda/2$ and $y = -\lambda/2$.
2. Substitute into the constraint: $x + y - 1 = -\lambda - 1 = 0$.
3. Therefore $\lambda = -1$ and $x = y = 1/2$.

Takeaway

λ tells us how sensitive the optimum is to the constraint $x + y = 1$. Relaxing the constraint by ε would change the optimum by roughly $-\lambda\,\varepsilon$.

Step 3: Introduce Parameters and Sensitivities

Consider a general design parameter s:

$$\min_{x} \; f(x, s) \quad \text{s.t.} \quad c(x, s) = 0.$$

- We already know how to find $x^{\star}(s)$.
- New question: how does the optimal value change as we change s?
- We want the sensitivity df/ds evaluated at the optimum.

Motivating Example

Heated rod: $-u''(x) = s(x)$ with $u(0) = u(1) = 0$, and cost $J = \int_0^1 (u - u_{\text{target}})^2 \, dx$.

Heated Rod Example

State (constraint):

$$-\frac{d^2u}{dx^2} = s(x), \qquad u(0) = u(1) = 0.$$

Objective:

$$J(s) = \int_0^1 \left(u(x; s) - u_{\text{target}}(x)\right)^2 dx.$$

- $s(x)$ controls heater strength along the rod.
- We seek $dJ/ds(x)$: "If I add heat here, how does the mismatch change?"

Naive Sensitivity Strategies

Finite Differences

Solve the PDE twice: $u(s)$ and $u(s+\varepsilon)$, then $dJ/ds \approx \frac{J(s+\varepsilon)-J(s)}{\varepsilon}$.

- Requires one extra PDE solve per component of s.
- Prohibitively expensive when s has many components.

Forward Sensitivity

Differentiate the constraint:

$$\frac{\partial c}{\partial u}\frac{du}{ds} + \frac{\partial c}{\partial s} = 0.$$

Solve for du/ds, then plug into $dJ/ds = \frac{\partial J}{\partial u}\frac{du}{ds} + \frac{\partial J}{\partial s}$.

- Still one large linear system **per component** of s.

Adjoint Method in Three Lines

First attempt (forward sensitivity):

$$c(u, s) = 0 \quad \Rightarrow \quad \frac{\partial c}{\partial u}\frac{du}{ds} + \frac{\partial c}{\partial s} = 0,$$

$$\frac{du}{ds} = -\left(\frac{\partial c}{\partial u}\right)^{-1}\frac{\partial c}{\partial s},$$

$$\frac{dJ}{ds} = \frac{\partial J}{\partial u}\frac{du}{ds} + \frac{\partial J}{\partial s}.$$

Issue: Need to solve this linear system *once per component* of s.

Adjoint Method in Three Lines

Adjoint trick (one solve total):

1. Form $\mathcal{L}(u, s, \lambda) = J(u, s) + \lambda^\top c(u, s)$.
2. Set $\partial \mathcal{L} / \partial u = 0$:

$$\frac{\partial J}{\partial u} + \lambda^\top \frac{\partial c}{\partial u} = 0 \quad \Rightarrow \quad \left(\frac{\partial c}{\partial u}\right)^\top \lambda = -\frac{\partial J}{\partial u}.$$

   Solve this once for λ; it depends on (u, s) but not on the dimension of s.

3. Stationarity in s gives the gradient:

$$\frac{dJ}{ds} = \frac{\partial J}{\partial s} + \lambda^\top \frac{\partial c}{\partial s}.$$

Result

Solve *one* adjoint system for λ and reuse it to obtain all components of dJ/ds.

Adjoint for the Heated Rod

- Constraint: $-\frac{d^2 u}{dx^2} = s(x)$.
- Objective: $J = \int (u - u_{\text{target}})^2 dx$.

Adjoint PDE (stationarity w.r.t. u):

$$-\frac{d^2\lambda}{dx^2} = -2(u - u_{\text{target}}), \qquad \lambda(0) = \lambda(1) = 0.$$

Gradient (stationarity in s):

$$\frac{dJ}{ds}(x) = -\lambda(x)$$

(we can flip the sign convention if we prefer to work with $+\lambda$).

Interpretation

$\lambda(x)$ tells us how helpful a unit of heat at x would be in reducing the cost.

Deriving the Heated-Rod Adjoint (1/2)

Constraint (state equation):

$$c(u, s) = -\frac{d^2 u}{dx^2} - s(x) = 0.$$

Objective:

$$J(u) = \int_0^1 (u - u_{\text{target}})^2 \, dx.$$

Form the Lagrangian

$$\mathcal{L}(u, s, \lambda) = J(u) + \int_0^1 \lambda(x) \, c(u, s) \, dx.$$

Stationarity with respect to u:

$$\frac{\partial \mathcal{L}}{\partial u} = 2(u - u_{\text{target}}) - \frac{d^2 \lambda}{dx^2} = 0.$$

Resulting adjoint PDE:

$$-\frac{d^2 \lambda}{dx^2} = -2(u - u_{\text{target}}), \quad \lambda(0) = \lambda(1) = 0.$$

Deriving the Heated-Rod Adjoint (2/2)

Stationarity with respect to s gives

$$\frac{dJ}{ds}(x) = \frac{\partial J}{\partial s} + \lambda(x)\frac{\partial c}{\partial s} = -\lambda(x).$$

Throughout these slides we adopt the sign convention where the reported gradient is $-\lambda(x)$ so that increasing s follows the negative adjoint direction.

Summary

1. Solve the forward problem $-u'' = s$.
2. Solve the adjoint problem $-\lambda'' = -2(u - u_{\text{target}})$.
3. Use $\frac{dJ}{ds}(x) = \lambda(x)$ to update heater strengths.

From Adjoint to Backpropagation

- Classical backpropagation = adjoint method for discrete layers.
- **Forward pass:** compute activations (state trajectory).
- **Backward pass:** propagate adjoint sensitivities (gradients) layer by layer.

Key Analogy

Weights w_i play the role of s_i. Layer activations are the state u. The backward pass solves the discrete adjoint system.

Neural ODE Adjoint in Practice

Forward dynamics:

$$\frac{dz}{dt} = f_\theta(z, t), \qquad z(t_0) = z_0.$$

Adjoint dynamics:

$$\frac{d\lambda}{dt} = -\lambda^\top \frac{\partial f_\theta}{\partial z}, \qquad \lambda(t_1) = \frac{\partial \mathcal{L}}{\partial z(t_1)}.$$

Parameter gradient:

$$\frac{\partial \mathcal{L}}{\partial \theta} = -\int_{t_0}^{t_1} \lambda^\top \frac{\partial f_\theta}{\partial \theta}\, dt.$$

Practical Pipeline

One forward ODE solve for $z(t)$ + one backward adjoint ODE solve for $\lambda(t)$ = gradients for *all* parameters with $\mathcal{O}(1)$ memory.

Algorithm 1: Adjoint Sensitivity Method

Input: Dynamics f, loss L, initial state $h(t_0)$, times $t_0 < t_1$

Forward Pass:

1. Solve ODE: $h(t_1) = h(t_0) + \int_{t_0}^{t_1} f(h(t), t, \theta)\, dt$
2. Compute loss: $L = L(h(t_1))$

Backward Pass: Define augmented state $s(t) = [\lambda(t), \frac{\partial L}{\partial \theta}(t), \frac{\partial L}{\partial t_0}(t)]$
Initialize: $\lambda(t_1) = \frac{\partial L}{\partial h(t_1)}$, others zero

Solve augmented ODE backward from t_1 to t_0:

$$\frac{d\lambda(t)}{dt} = -\lambda(t)^T \frac{\partial f(h(t), t, \theta)}{\partial h}$$
$$\frac{d}{dt}\frac{\partial L}{\partial \theta} = -\lambda(t)^T \frac{\partial f(h(t), t, \theta)}{\partial \theta}$$
$$\frac{d}{dt}\frac{\partial L}{\partial t_0} = -\lambda(t)^T f(h(t), t, \theta)$$

The Flow Map: Integral Form of Neural ODE

Definition: The flow map Φ integrates the ODE from t_0 to t_1:

$$z(t_1) = \Phi(z(t_0), t_0, t_1, \theta) = z(t_0) + \int_{t_0}^{t_1} f(z(\tau), \tau, \theta)\, d\tau \tag{14}$$

Key Properties:

- Φ is the exact solution operator of the ODE
- Depends on initial condition $z(t_0)$, time interval, and parameters θ
- Computed numerically via ODE solver (Euler, RK4, etc.)
- Differentiable with respect to all inputs!

The Loss Function:

$$L = L(z(t_1)) = L(\Phi(z(t_0), t_0, t_1, \theta)) \tag{15}$$

The Training Problem: Compute $\frac{dL}{d\theta}$ efficiently

The Gradient Challenge

What we want: $\frac{dL}{d\theta}$ where $L = L(\Phi(z(t_0), t_0, t_1, \theta))$

Chain rule attempt:

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial z(t_1)} \frac{\partial z(t_1)}{\partial \theta} \tag{16}$$

The Problem: Computing $\frac{\partial z(t_1)}{\partial \theta}$ requires tracking how parameters affect the entire trajectory!

$$\frac{\partial z(t_1)}{\partial \theta} = \frac{\partial}{\partial \theta} \left[z(t_0) + \int_{t_0}^{t_1} f(z(\tau), \tau, \theta) d\tau \right] \tag{17}$$

This requires $\frac{\partial z(\tau)}{\partial \theta}$ for all $\tau \in [t_0, t_1]$ (the hidden states!)

The Core Issue

Can't pull $\frac{\partial}{\partial \theta}$ inside the integral because $z(\tau)$ itself depends on θ!

Lagrange Multipliers: The Setup

Constrained Optimization: Minimize $L(z(t_1))$ subject to $\frac{dz}{dt} = f(z, t, \theta)$

Key Idea: Turn constraint into penalty using Lagrange multiplier $\lambda(t)$

$$\mathcal{L} = L(z(t_1)) - \int_{t_0}^{t_1} \lambda(t)^T \left[\frac{dz}{dt} - f(z(t), t, \theta) \right] dt \tag{18}$$

Why This Helps:

- When constraint is satisfied: $\frac{dz}{dt} = f$ so integral $= 0$
- Thus: $\mathcal{L} = L(z(t_1))$ (Lagrangian equals original loss)
- So: $\frac{d\mathcal{L}}{d\theta} = \frac{dL}{d\theta}$ (what we want!)
- But: We get to choose $\lambda(t)$ strategically!

The Strategy

Choose $\lambda(t)$ to eliminate the problematic $\frac{\partial z(\tau)}{\partial \theta}$ terms

Deriving the Adjoint Equation (Step 1)

Expand the Lagrangian:

$$\mathcal{L} = L(z(t_1)) - \int_{t_0}^{t_1} \lambda(t)^T \frac{dz}{dt} dt + \int_{t_0}^{t_1} \lambda(t)^T f(z(t), t, \theta) dt \tag{19}$$

Integration by Parts on the middle term:

$$\int_{t_0}^{t_1} \lambda(t)^T \frac{dz}{dt} dt = \lambda(t_1)^T z(t_1) - \lambda(t_0)^T z(t_0) - \int_{t_0}^{t_1} \frac{d\lambda}{dt}^T z(t) dt \tag{20}$$

Substitute back:

$$\mathcal{L} = L(z(t_1)) - \lambda(t_1)^T z(t_1) + \lambda(t_0)^T z(t_0) \tag{21}$$

$$+ \int_{t_0}^{t_1} \left[\frac{d\lambda}{dt}^T z(t) + \lambda(t)^T f(z(t), t, \theta) \right] dt \tag{22}$$

Deriving the Adjoint Equation (Step 2)

Take derivative with respect to θ:

$$\frac{d\mathcal{L}}{d\theta} = \frac{\partial L}{\partial z(t_1)}\frac{\partial z(t_1)}{\partial \theta} - \lambda(t_1)^T\frac{\partial z(t_1)}{\partial \theta} + \lambda(t_0)^T\frac{\partial z(t_0)}{\partial \theta} \tag{23}$$

$$+ \int_{t_0}^{t_1}\left[\frac{d\lambda}{dt}^T\frac{\partial z}{\partial \theta} + \lambda(t)^T\left(\frac{\partial f}{\partial z}\frac{\partial z}{\partial \theta} + \frac{\partial f}{\partial \theta}\right)\right]dt \tag{24}$$

Group terms with $\frac{\partial z}{\partial \theta}$:

$$\frac{d\mathcal{L}}{d\theta} = \left(\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)^T\right)\frac{\partial z(t_1)}{\partial \theta} \tag{25}$$

$$+ \int_{t_0}^{t_1}\left[\left(\frac{d\lambda}{dt}^T + \lambda(t)^T\frac{\partial f}{\partial z}\right)\frac{\partial z}{\partial \theta}\right]dt \tag{26}$$

$$+ \int_{t_0}^{t_1}\lambda(t)^T\frac{\partial f}{\partial \theta}dt \tag{27}$$

Note: $z(t_0)$ is fixed, so $\frac{\partial z(t_0)}{\partial \theta} = 0$

The Adjoint Solution: Making Terms Vanish

Choose $\lambda(t)$ to eliminate all $\frac{\partial z}{\partial \theta}$ terms!

Boundary condition at t_1:

$$\lambda(t_1)^T = \frac{\partial L}{\partial z(t_1)} \quad \Rightarrow \quad \text{Boundary term vanishes!} \tag{28}$$

Dynamics of $\lambda(t)$ (adjoint equation):

$$\frac{d\lambda}{dt}^T = -\lambda(t)^T \frac{\partial f}{\partial z} \quad \Rightarrow \quad \text{Integral term vanishes!} \tag{29}$$

Final Gradient (what remains):

$$\frac{dL}{d\theta} = \int_{t_0}^{t_1} \lambda(t)^T \frac{\partial f}{\partial \theta} dt = -\int_{t_1}^{t_0} \lambda(t)^T \frac{\partial f}{\partial \theta} dt \tag{30}$$

Key Result

$\lambda(t) =$ adjoint state, solve backward from t_1 to t_0, then compute gradient!

How Neural ODEs Automate This

The Traditional Way (Impossible):

1. Derive adjoint equations by hand for your specific f
2. Compute Jacobians $\frac{\partial f}{\partial z}$ and $\frac{\partial f}{\partial \theta}$ analytically
3. Implement custom backward pass

The Neural ODE Way (Automatic):

1. Define $f(z, t, \theta)$ as a neural network in PyTorch/JAX
2. **Autodiff gives you $\frac{\partial f}{\partial z}$ and $\frac{\partial f}{\partial \theta}$ for FREE!**
3. Solve adjoint ODE using same ODE solver, but backward

The Magic of Autodiff

Vector-Jacobian Products (VJPs):
$\lambda(t)^T \frac{\partial f}{\partial z}$ and $\lambda(t)^T \frac{\partial f}{\partial \theta}$
computed via reverse-mode autodiff without forming full Jacobian!

Complexity: VJP costs $\approx$ 2-3$\times$ forward pass (not O(d²) for Jacobian!)

Adjoint Method: Why $\mathcal{O}(1)$ Memory?

Standard Backprop through ODE Solver:

- Store all intermediate states $h(t_i)$ for $i = 1, \ldots, N$
- N depends on adaptive step size (could be 100s or 1000s)
- Memory: $\mathcal{O}(N)$ where $N =$ number of function evaluations

Adjoint Method:

- Only store final state $h(t_1)$
- During backward pass, recompute $h(t)$ as needed
- Memory: $\mathcal{O}(1)$ – just the current state!

Trade-off

Memory $\mathcal{O}(1)$ but computation $\approx 2\times$ (one forward, one backward solve)

Hyperparameter Selection

ODE Solver Tolerance

- `rtol`, `atol`: Control accuracy
- Higher tolerance $\rightarrow$ faster but less accurate
- Typical: `rtol=1e-3`, `atol=1e-4`

Solver Method

- **Adaptive**: 'dopri5', 'adams' (recommended)
- **Fixed-step**: 'euler', 'rk4' (for debugging)

Integration Time

- Usually $T = 1.0$ (can be learned)
- Longer $T \rightarrow$ more expressive but slower

Latent ODEs for Irregular Time Series

Challenge: Irregular, sparse observations with variable time gaps

Approach: Combine RNNs and ODEs

- **Encoder (ODE-RNN):** Process observations backward in time
- **Latent ODE:** Smooth dynamics in continuous time
- **Decoder:** Generate predictions at any time

Key Innovation: Poisson process prior for observation times

$$p(t_1, \ldots, t_N | z_0) = \prod_{i=1}^{N} \lambda(t_i | z_0) \exp\left(-\int_0^T \lambda(t|z_0) dt\right) \tag{31}$$

Models *when* observations occur, not just their values.

Latent ODE Architecture Details

Three-Component System

1. ODE-RNN Encoder (Backward)

Process observations $\{(t_i, x_i)\}_{i=1}^{N}$ in *reverse* time order:

$$h_i = \text{ODESolve}(h_{i+1}, t_{i+1} \to t_i) \quad \text{then} \quad h_i \leftarrow \text{RNN}(h_i, x_i)$$

Output: Initial latent state $z_0 \sim q(z_0 | x_{1:N})$

2. Latent ODE Dynamics

Continuous evolution in latent space:

$$\frac{dz(t)}{dt} = f_\theta(z(t), t)$$

3. Decoder

Map latent states to observations: $p(x_i | z(t_i))$

1. **Why Process Backward?** This is a key design choice! Processing observations backward in time naturally produces an initial condition z_0 that encodes the entire sequence. Think of it like reverse-engineering the initial state from the trajectory.
   The ODE-RNN Step: At each observation time t_i (going backward):
   1.1 Evolve hidden state from t_{i+1} to t_i using an ODE: this accounts for the time gap
   1.2 Update with RNN cell using observation x_i: this incorporates the data

   This is different from standard RNNs which assume fixed time steps. The ODE naturally handles variable gaps!
   Why This Architecture Works:
   - Encoder handles irregularity by explicitly modeling time via ODEs
   - Latent space has smooth, continuous dynamics (good inductive bias)
   - Can query $z(t)$ at any time, not just observation points
   - Decoder can make predictions at arbitrary future times

Training Latent ODEs: The ELBO

Evidence Lower Bound (ELBO): Variational inference objective

$$\mathcal{L}_{\text{ELBO}} = \underbrace{\mathbb{E}_{q(z_0|x)}[\sum_{i=1}^{N} \log p(x_i|z(t_i))]}_{\text{Reconstruction}} - \underbrace{D_{KL}(q(z_0|x)\|p(z_0))}_{\text{Regularization}} \tag{32}$$

Component 1: Reconstruction Loss

- Measures how well the model predicts observations
- $q(z_0|x)$: Encoder's posterior over initial state
- $z(t_i)$: Latent state at time t_i via ODE
- Typically Gaussian: $\log p(x_i|z(t_i)) = \log \mathcal{N}(x_i|\mu(z(t_i)), \sigma^2)$

Component 2: KL Divergence

- Regularizes latent space to match prior $p(z_0) = \mathcal{N}(0, I)$
- Prevents overfitting and ensures smooth latent space
- Only computed at $t = 0$, not entire trajectory!

Latent ODE: Handling Observation Times

Innovation: Model *when* observations occur, not just their values

Poisson Process Intensity:

$$\lambda(t|z_0) = g_\psi(z(t)) \tag{33}$$

where g_ψ is a neural network mapping latent states to observation rates.

Joint Likelihood:

$$p(\{x_i, t_i\}_{i=1}^N|z_0) = \underbrace{\prod_{i=1}^N p(x_i|z(t_i))}_{\text{observations}} \times \underbrace{\prod_{i=1}^N \lambda(t_i|z_0) \exp\left(-\int_0^T \lambda(t|z_0)dt\right)}_{\text{timing}} \tag{34}$$

The integral $\int_0^T \lambda(t|z_0)dt$ is computed by solving an ODE!

Function Encoders with Neural ODEs

Goal: Transfer learned dynamics to new systems without gradient updates

Approach: Learn a basis of dynamics

1. Learn K basis ODEs: $\frac{dz_i}{dt} = f_i(z_i, t)$ for $i = 1, \ldots, K$
2. For new system: encode demonstrations $\rightarrow$ coefficients α_i
3. Predict: $\frac{dz}{dt} = \sum_{i=1}^{K} \alpha_i f_i(z, t)$

Key Idea: Treat dynamics as vectors in a Hilbert space
Any trajectory $x(z, t)$ can be represented as: $x \approx \sum_{i=1}^{K} \alpha_i \phi_i$

Zero-Shot Transfer

Compute coefficients α_i from demonstrations without any gradient updates!

1. **The Big Picture:** This is a fundamentally different approach to transfer learning. Instead of fine-tuning a model for each new task, we learn a *dictionary of dynamics* that can be combined to represent new systems.

   Analogy to Fourier Series: Any periodic function can be written as $f(x) = \sum_k a_k \sin(kx) + b_k \cos(kx)$. The sine and cosine functions form a basis. Given any function f, we can compute coefficients a_k, b_k via integrals (inner products). We're doing the same thing, but with dynamical systems instead of functions!

   Why Neural ODEs? Each basis element ϕ_i is itself a Neural ODE with dynamics f_i. During training, we learn multiple Neural ODEs simultaneously (like learning both sines and cosines). The training objective encourages the basis to span a diverse space of dynamics.

   Training Phase:
   - Train on multiple dynamical systems (e.g., different robot configurations, different physical parameters)
   - Each system has multiple demonstration trajectories
   - Learn basis functions $\phi_1, \ldots, \phi_K$ that can represent all training systems as linear combinations

   Test Phase (Zero-Shot):
   - Given a NEW system (never seen during training)
   - Observe a few demonstration trajectories

Computing Demonstration Velocity: Given demonstration points $\{(t_j, z_j)\}$:

- **Direct differentiation:** If demonstration is a continuous function, compute $\frac{dz}{dt}$ numerically
- **Finite differences:** For discrete observations: $v(t_j) \approx \frac{z_{j+1}-z_j}{t_{j+1}-t_j}$

Inner Product Formula:

$$\alpha_i = \mathbb{E}_{t\sim\mathcal{U}[0,T],z\sim z_i(t)}\Big[\underbrace{v_{\text{demo}}(z,t)}_{\text{demo velocity}} \cdot \underbrace{f_i(z,t)}_{\text{basis velocity}}\Big] \tag{35}$$

Key Properties:

- $\alpha_i > 0$: demonstration aligns with basis i
- $\alpha_i < 0$: demonstration opposes basis i
- $|\alpha_i|$ large: basis i is important for this system

Extensions

Augmented Neural ODEs

Add extra dimensions to avoid topological constraints

Second-order Neural ODEs

Include acceleration: $\frac{d^2h}{dt^2} = f(h, \frac{dh}{dt}, t)$

Stochastic Differential Equations (SDEs)

Add noise for uncertainty: $dh = f(h, t)dt + g(h, t)dW$

Hamiltonian Neural Networks

Preserve energy and symplectic structure

Practical Tip: Neural ODEs + Interpretable Models

Problem: Neural ODEs are powerful but not interpretable
$f(x, t, \theta)$ is a black-box neural network – can't extract equations!

Solution: Use Neural ODEs as a preprocessing step

1. Train Neural ODE on irregular/noisy data
2. Generate clean, regularly-spaced data from trained Neural ODE
3. Pass regular data to interpretable method (SINDy, symbolic regression)

Best of Both Worlds

Neural ODE: handles irregular data, noise robust, accurate
SINDy/Symbolic: gives interpretable equations like $\dot{x} = \mu x - x^3$

Summary

Key Takeaways

1. Neural ODEs = continuous-depth neural networks
2. ResNets are crude Euler integration; Neural ODEs use better solvers
3. Adjoint method enables $\mathcal{O}(1)$ memory training via autodiff
4. Works with irregular time series (unlike ResNets/RNNs)
5. Can bake in physical structure (Hamiltonian, Lagrangian, symplectic)
6. Applications: classification, time series, generative models, physics

The Big Idea

Learn the **vector field** (continuous dynamics), not discrete transformations

Key advantage: Leverage 300+ years of ODE theory and numerical methods!

References

Chen, R. T. Q., Rubanova, Y., Bettencourt, J., & Duvenaud, D. (2018).
Neural Ordinary Differential Equations.
NeurIPS 2018 (Best Paper Award).

Grathwohl, W., Chen, R. T. Q., Bettencourt, J., Sutskever, I., & Duvenaud, D. (2019).
FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models.
ICLR 2019.

Rubanova, Y., Chen, R. T. Q., & Duvenaud, D. (2019).
Latent ODEs for Irregularly-Sampled Time Series.
NeurIPS 2019.